GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory

KAREL ADÁMEK, Oxford e-Research Centre, Department of Engineering Science, University of Oxford, United Kingdom
SOFIA DIMOUDI, Centre for Advanced Instrumentation, Durham University, United Kingdom
MIKE GILES, Mathematical Institute, University of Oxford, United Kingdom
WESLEY ARMOUR, Oxford e-Research Centre, Department of Engineering Science, University of Oxford, United Kingdom

We present an implementation of the overlap-and-save method, a method for the convolution of very long signals with short response functions, which is tailored to GPUs. We have implemented several FFT algorithms (using the CUDA programming language), which exploit GPU shared memory, allowing for GPU accelerated convolution. We compare our implementation with an implementation of the overlap-and-save algorithm utilizing the NVIDIA FFT library (cuFFT). We demonstrate that by using a shared-memory-based FFT, we can achieved significant speed-ups for certain problem sizes and lower the memory requirements of the overlap-and-save method on GPUs.

CCS Concepts: • **Applied computing;** • **Computing methodologies** → *Parallel computing methodologies*;

Additional Key Words and Phrases: Fast convolution, CUDA, GPU, overlap-and-save, FFT

ACM Reference format:
Karel Adámek, Sofia Dimoudi, Mike Giles, and Wesley Armour. 2020. GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory. *ACM Trans. Archit. Code Optim.* 17, 3, Article 18 (August 2020), 20 pages.
https://doi.org/10.1145/3394116

1 INTRODUCTION

Convolution is one of the most fundamental signal filtering techniques, widely used in signal processing, to aid discovery in many areas of natural sciences. It is a linear operation involving an input signal s of length N_s and a response function (or a filter) h of length M. There are two principal approaches to linear filtering, where their usability depends on the length of the response function h.

This work has received support from STFC Grant No. ST/R000557/1. This work is also supported by a Leverhulme Trust Project Grant (ARTEMIS: Real-time discovery in Radio Astronomy).
Authors' addresses: K. Adámek and W. Armour (corresponding author), Oxford e-Research Centre, Department of Engineering Science, University of Oxford, 7 Keble Road, Oxford, OX1 3QG, United Kingdom; emails: karel.adamek@gmail.com, wes.armour@oerc.ox.ac.uk; S. Dimoudi, Centre for Advanced Instrumentation, Department of Physics, Science Laboratories, South Road, Durham, DH1 3LE, United Kingdom; email: sofia.dimoudi@durham.ac.uk; M. Giles, Mathematical Institute, University of Oxford, Andrew Wiles Building, Radcliffe Observatory Quarter (550), Woodstock Road, Oxford, OX2 6GG, United Kingdom; email: mike.giles@maths.ox.ac.uk.

1544-3566/2020/08-ART18
https://doi.org/10.1145/3394116

When the filter (h) is short it might be beneficial to calculate convolution in the time-domain using the formula for discrete convolution,

$$y[n] = h[k] \star s[n] = \sum_{k=0}^{M-1} s[n-k]h[k], \tag{1}$$

where $y[n]$ are elements of the filtered signal, and brackets [ ] denote quantities that are discrete (sampled). The complexity of time-domain convolution is $O\left(N_s^2\right)$.

If we have a longer filter, then it might be better to invoke the convolution theorem and calculate convolution in the frequency-domain using a Fourier transformation. The convolution theorem states that [14]

$$h[k] \star s[n] = \mathrm{FT}^{-1}\left(H[m] \cdot S[m]\right), \tag{2}$$

where $H = \mathrm{FT}(h)$ and $S = \mathrm{FT}(s)$ are Fourier pairs of h and s and FT and FT^{-1} is discrete Fourier transformation and its inverse, respectively. By using Fourier transformation in the convolution calculation, we are performing circular convolution (as opposed to linear convolution (Equation (1))), which introduces an aliasing effect, where samples at the edges[1] of the input signal are added together rendering them useless for convolution. Therefore, we have to pad both the filter and the input signal with zeros (called zero padding), to the same size of at least $0 \leq m < N_s + M - 1$.

The convolution theorem allows us to replace convolution in the time-domain by point-wise multiplication in the frequency-domain. This, however, would not be computationally feasible without the Fast Fourier Transformation (FFT) algorithm, which decreases the cost of the discrete Fourier transformation to $O(N_s \log_2(N_s))$. Using the FFT algorithm and the convolution theorem to perform convolutions is often called *fast convolution*.

Determining when to use time-domain convolution as opposed to frequency-domain convolution depends on many factors including the character of the problem being solved, implementation, the hardware used, and so on.

As mentioned above, frequency-domain convolution requires that the input signal and the filter are both of the same length. To calculate the convolution of a long input signal in the frequency-domain, we have to perform long FFTs on both. This can be very inefficient in terms of computations and memory storage, particularly if we are applying multiple filters. Two commonly used algorithms to overcome these shortcomings are the *overlap-and-save* (OLS) or *overlap-and-add* (OLA) [19] methods.

The overlap-and-save (add) is a hybrid method that combines advantages of time-domain convolution with frequency-domain convolution. It allows us to break the input signal into segments of length N and use fast convolution independently on each segment. The two methods differ in the way they deal with aliased samples and how the output is constructed. The overlap-and-save method discards the aliased samples from each segment and saves only the correct part of the segment to an appropriate place in the output signal. The overlap-and-add method adds together aliased samples from the neighboring segments to create the correct output. Therefore a parallel implementation of the overlap-and-add method requires exclusive access to the areas of memory that contain the aliased output signal.

The fast convolution, which is performed on each segment, has four steps: forward FFT of a segment; point-wise complex multiplication of the filter and the segment in frequency-domain;

[1]This depends on the character of the filter used. Filters that use only future samples will be aliased with the end of the segment, filters that use past samples will be aliased with the beginning of the segment, while a time-centred filter introduces aliasing at both ends. The number of aliased samples is equal to the unpadded length M of the filter.

inverse FFT of the convolved segment; and rejection of the edges. These steps are traditionally performed using libraries or custom code, with the input and output stored in the GPU device memory[2] for each step. This is a limiting factor when considering the convolution of the segment as a whole.

The novelty of this work and its focus is to enable fast convolution by storing signal segments and filters in the fastest areas of GPU memory. Performing the convolution and the associated inverse FFT on data held in these fast memories allows us to eliminate device memory traffic and hence accelerate the convolution algorithm on GPUs.

The novelty of this work and its focus is to enable fast convolution by exploiting the fastest areas of GPU memory, registers and shared memory. To do this, we needed to write FFT codes that will operate directly on data stored in shared memory (NVIDIA library functions do not do this). Using these codes, we are able to perform the convolution and the associated forward and inverse FFT on data held in the fastest areas of GPU memory and hence accelerate the convolution algorithm on GPUs. Specifically, we can eliminate expensive access to the device (global) memory, which is otherwise required. With this goal in mind, we have implemented a basic version of the Cooley-Tukey FFT algorithm [11] for complex-to-complex FFTs and a basic version of the Stockham FFT algorithm [3] for real-to-complex and complex-to-real FFTs. We have implemented these FFT algorithms so that they can execute on data held in shared memory.[3] The purpose of this work is to demonstrate the viability of our approach of moving operations into GPU kernels using device ready algorithms. The choice of the optimal FFT algorithm and implementation of optimized and efficient FFT algorithms on GPUs is beyond the scope of this work but will serve as a focus of our future work.

We have chosen to focus only on the overlap-and-save method rather than on the overlap-and-add method, because the overlap-and-add method would require a synchronization step between segments due to a race condition that would occur when neighbouring segments try to write their computed data to the output signal stored in GPU device memory.

The work presented in this article was developed for NVIDIA GPUs; therefore, we have used the CUDA language extension for our work. The investigation of OpenCL or any other framework is outside the scope of this work. This work has been used to enable real-time processing of time-domain radio astronomy data [1, 4, 5].

Our GPU implementation of the overlap-and-save method with a basic user interface is available on GitHub.[4] The user interface we provide allows the user to test the functionality of our implementation. A more detailed description is provided on our Github wiki.

2 RELATED WORK

The comprehensive study of the convolution algorithms on CPUs, GPUs, and FPGAs was conducted by Fowers et al. [8]. They have compared convolution algorithms by their computational cost, energy efficiency and execution time for a range of input signal sizes and filter lengths. Their investigation shows that the time-domain convolution is faster for either short filters or short input signals. For longer input signals and longer filters, it is beneficial to use the overlap-and-save method. The performance of the NVIDIA cuDNN library, in the context of convolutional neural networks, was investigated by Jordà et al. [12]. The authors present different algorithms used by

[2]Device memory (sometimes called main memory or global memory) has the lowest memory bandwidth on the GPU and as such takes the most time to access.
[3]Shared memory is a small but fast area of GPU memory and can be treated as a user managed cache.
[4]https://github.com/KAdamek/GPU_Overlap-and-save_convolution.

the cuDNN to calculate two-dimensional convolution. Although this is for two-dimensional convolutions it shows the advantage of frequency-domain convolution for larger filters and input signals.

Both overlap-and-save (or OLA) and FFT algorithms are well known and extensively researched, having lots of coverage in literature. Both OLS and OLA methods have been implemented on GPUs [6, 13]. The theory of these methods is also actively developed, for example [7, 16, 25], and references within.

The FFT algorithm and its implementation on GPUs is equally well researched and extensive publications can be found on the subject, for example [9, 10, 15, 22, 23, 26]. Govindaraju et al. [9] focused on providing a set of FFT routines that would be applicable to a wide range of input signal lengths. The authors have used the Stockham algorithm to avoid reordering of the elements, which is required when the Cooley-Tukey algorithm is used. Gutierrez et al. [10] deals with longer FFT from the host perspective with emphasis on long input signals. They have implemented the decimation-in-time Cooley-Tukey algorithm where part of the FFT is performed in shared memory and Moreland and Angel [15] described the implementation of the two-dimensional FFT real-to-real algorithm for image processing. More on FFTs in general can be found in Ref. [21].

There is also a number of GPU FFT source codes available [22, 23, 26]. However, these FFT codes were not suited for our needs for integration into the overlap-and-save method. The primary reason for this is that these FFT codes were not designed as device callable functions.

The FFT by Volkov and Kazian [23] stores larger FFTs (16 elements or more) using thread registers. Our implementation of convolution uses registers to store the values of the signal segment and current filter value. Further register utilization would lead to code slowdown.

The FFT code by Vasilache et al. [22] focuses on FFT lengths that are too small for our intentions. The FFT length considered in the article is $N < 256$. We require our implementation to work with the largest filters permitted by either shared memory[5] or the number of active threads per thread-block. For example a filter size of 512 elements would require an FFT length of at least 1,024 elements or longer.

Last, the FFT code by Yang and Zhou [26] was written for the Fermi generation of GPUs and has not been updated for more modern GPU architectures.

Our FFT implementation differs from the previously published works, because it is designed to use shared memory only and to be called from the GPU kernel itself. Therefore, it deals only with short FFT lengths due to size limitation of the shared memory (currently $N \leq 4{,}096$) and where N is a power of two. Moreover, our implementation of the Cooley-Tukey FFT algorithm cannot be used as a standalone FFT routine as it lacks element reordering, which is not required for calculation of the convolution.

3 IMPLEMENTATION

We present our implementation of the overlap-and-save (OLS) method for NVIDIA GPUs using the CUDA programming language, which uses a shared-memory implementation of standard FFT algorithms to calculate one-dimensional convolutions. Our implementation of the OLS method can calculate complex-to-complex[6] (C2C) and real-to-real (R2R) convolutions. These implementations are compared to an implementation of (direct) convolution that uses the NVIDIA cuDNN library and also to an implementation of the OLS method, which uses the NVIDIA cuFFT library to perform the FFT parts of the OLS algorithm on the GPU.

In this section, we describe all implementations used in this article starting with the NVIDIA cuDNN library [17] implementation of convolution. Next, we describe the overlap-and-save

[5]The size of shared memory ultimately limits the size of a signal segment that can be processed in our method.
[6]Depending on the post-processing step this might be the complex-to-real convolution as well.

method and its implementation using the NVIDIA cuFFT library [18] (cuFFT OLS), which contains highly optimized and GPU ported FFT algorithms. Our implementation of the OLS method with shared-memory FFT (SM-OLS) is described last.

3.1 Convolution via NVIDIA cuDNN

The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of deep neural networks primitives. The cuDNN library offers (among many other routines) forward convolution, which we have used as a comparison.

Our cuDNN convolution implementation is a real-to-real. The cuDNN library uses a range of different algorithms based on the task and the size of the input. We have left the cuDDN library to chose the most suitable convolution algorithm for our test case by using the flag `CUDNN_CONVOLUTION_FWD_PREFER_FASTEST`. Our tests are performed with one-dimensional data with a single channel[7]; therefore, we have used the `CUDNN_TENSOR_NCHW` data layout. Since we cannot be sure how many operations are performed by the cuDNN library, we have not calculated the number of FLOPS for the cuDNN convolution implementation in our comparisons; instead, we use the number of processed elements per second.

3.2 Overlap-and-save Method

We will first describe the common steps of the OLS method, which are performed by all implementations. These steps apply to both C2C and R2R convolutions, since both are performed in the Fourier domain, which is complex.

A flow diagram of the overlap-and-save algorithm is shown in Figure 1 and the method is represented pictorially in Figure 2. We begin by separating the input signal of size S into N_{seg} independent segments, all subsequent operations are then applied independently on each and every segment. Next a forward FFT is applied to each segment. What follows is the frequency domain convolution of the segment A with every filter f from N_{fil} filters, that is complex multiplication of the segment with one or more filters. After that, we apply an inverse FFT to the results and then discard the aliased edges of each block, recombining the samples from all blocks into the output. Optionally, we can apply some post-processing to the resulting output. In essence, this operation transforms a blocked circular convolution into one that is linear and continuous.

In the overlap-and-save technique (shown in Figure 2), the length of the segment, that is the FFT length, N must be chosen such that it minimizes the fraction of discarded samples compared to the segment length. The number of discarded samples depends on the filter length M that is being applied to the signal and are equal to $M - 1$. Thus, the number of correct (unaliased) samples in the segment is $L = N - M + 1$. A higher fraction of discarded samples increases the overall number of segments required by the OLS method. To ensure good performance of the FFT algorithm on a signal segment, we limit the segment length N to be lengths equal to powers of two. The lengths of the segments from which we combine the convolved signal can be different for each implementation. The cuFFT-OLS performs better with longer segments while SM-OLS performs better with a shorter segment length. The convolved signal is not affected by the choice of the segment size.

3.3 OLS Method Using cuFFT Library

Using the cuFFT library, we have implemented one-dimensional convolution via the OLS method (cuFFT-OLS) for two variants of input data. We have implemented complex-to-complex and real-to-real convolutions. The pseudo-code for both variants of the cuFFT-OLS is shown in Algorithm 1.

[7]Channels in the context of cuDNN library are equivalent to the number of elements per structure in array-of-structures vs. structure-of-arrays data layouts. Since we have a simple data layout, we have used the equivalent of structure-of-arrays.

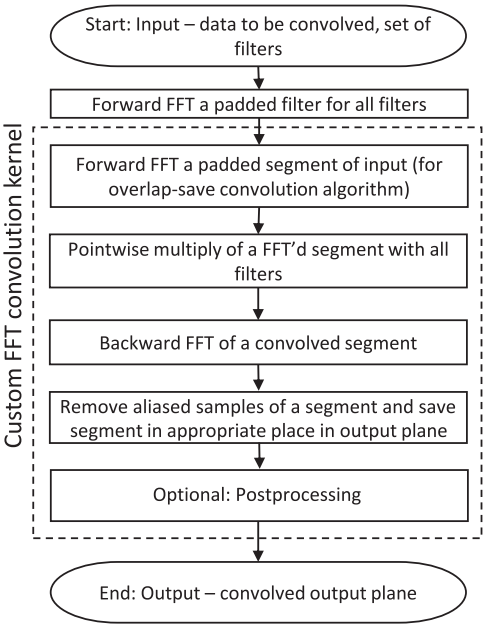

Fig. 1. Flow diagram of the overlap-and-save method: Our input is a signal that is to be convolved with a set of filters. The first step is to Fourier transform the padded filters. These will be used for convolution with each segment. In the next step, we separate the input signal into independent overlapping segments. The total overlap length for each segment is equal to the filter length, these segments need to be Fourier transformed. The third step is convolution in the form of complex point-wise multiplication. The convolved segment is then inverse Fourier transformed. In the last step, we remove the aliased part of each segment and merge the clean parts to produce a continuous output. Optionally, we can perform a post-processing step at the end.

These variants only differ in the type of FFT used for the forward and the backward Fourier transform. Both using the cuFFT library to perform FFT routines.

The most efficient way to implement cuFFT-OLS is to utilize a feature of the cuFFT library called callbacks. The cuFFT callbacks allow the user a per-element access to the data that are loaded or stored by the cuFFT routine and allow the user to perform pre- or post- processing of the data without any additional GPU kernels.

We can use callbacks together with the forward FFT to perform frequency domain convolution (complex multiplication of the sample segment with the appropriate sample from multiple filters) and also with the inverse FFT, where we can remove the aliased samples from the segment. While the latter eliminates problematic global memory access, the former callback has less effect.

The callback used together with the inverse FFT means we do not need to store intermediate segments with aliased samples into the device memory. This is a significant bandwidth saving, since the intermediate result is of size $N_{\text{seg}}SF$ and it would have to be written to main memory (the output from cuFFT), then read so that aliased samples can be removed, then sorted as the final (corrected) output.

The forward FFT callback eliminates proportionally only a small device memory access to the segments after the forward FFT. The main device memory access, which stores the result of the frequency domain multiplication, remains intact. Therefore, the impact of this callback is marginal for a large number of filters.

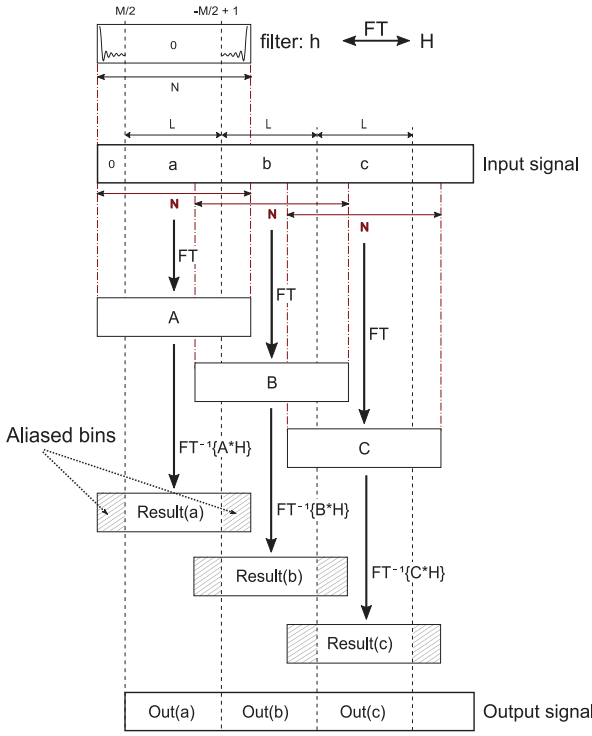

Fig. 2. Overlap-and-save method: The input signal, of length N_s, is separated into overlapping segments (A,B,C,...), where the amount of overlap is given by the filter length M. These segments are then processed independently, where FT denotes Fourier transformation. At the end the aliased samples of the segment, which are equal to the filter length are discarded. This example uses a time-centred filter, which aliases both ends of the segment.

The cuFFT library also allows the user to use some shared memory. The amount is however limited to 16 kB, which can accommodate only 2,048 FFT elements, while the optimal FFT length for cuFFT library is 8,192 elements. Furthermore, this does not allow us to use forward and backward transform and as such does not remove problematic device memory access.

The disadvantages of the cuFFT-OLS implementation are that it has to load and store intermediate data to the device memory in between the frequency domain convolution (forward FFT step) and the inverse FFT. Another disadvantage is higher memory requirements as the last step (where we remove the aliased samples of the segments) cannot be performed in-place due to the non-deterministic nature of thread-block scheduling on GPUs. The advantage of the cuFFT implementation is that it works for any filter length and only relies on NVIDIA supported libraries.

3.4 OLS Method Using Shared-memory FFT

We present two versions of the one-dimensional overlap-and-save (OLS) method, which is performed in the shared memory for NVIDIA GPUs using the CUDA programming language. The first implementation of OLS is for complex-to-complex[8] (C2C) convolutions, using a shared-memory implementation of the Cooley-Tukey [11] FFT algorithm. The second implementation of the OLS method is for real-to-real (R2R) convolutions. This implementation uses a shared-memory

[8]Depending on the post-processing step this might be complex-to-real convolution as well.

ALGORITHM 1: Pseudo-code for the cuFFT-OLS implementation. For the input, we have input data x and set of filters f. The output is the convolved result y. The FFT routines (`ForwardFFT` and `InverseFFT`) are either C2C for the complex input or R2C and C2R, respectively, for the real input.

Input: x, f;
Output: y;
Forward FFT of the filters;
F = `ForwardFFT` (f);
Separation of the signal into segments a, b, c, …;
(a, b, c, …) = `Separate` (x);
Forward FFT of the individual segments;
(A, B, C, …) = `ForwardFFT` (a, b, c, …);
Callback begin
    Per-element complex multiplication of the segment a with N_{fil} filters;
    for $s = 0$ **to** N_{seg} **do**
        for $r = 0$ **to** N_{fil} **do**
            $A[s] = A[s] \times F[r][s]$;
        end
    end
end
(a, b, c, …) = `InverseFFT` (A, B, C, …);
Callback begin
    y = `RemoveAliasedSamples` (a, b, c, …);
end

implementation of the Stockham FFT algorithm [3]. Our shared-memory implementation of the OLS method follows the same steps as the cuFFT-OLS implementation, but has a significant difference, it incorporates all the steps required by the OLS method into one GPU kernel. This is possible because we can call forward and inverse FFT device functions directly from the GPU kernel, which eliminates the computationally costly device memory transactions, working instead on data held in shared memory and GPU registers. The pseudo-code for our shared-memory OLS method is presented in Algorithm 2.

In our implementation of convolution through the OLS method in shared memory, each thread-block[9] is assigned to one segment of the input data. Each thread-block applies a shared-memory forward FFT and stores segment samples, which are now in the frequency domain, into registers. Each thread from the thread-block works with four samples. These segment samples are reused throughout the execution of the thread-block. Stored segment samples are then complex multiplied with appropriate samples from one or more filters. These filters are already in the frequency domain, since they were Fourier transformed before thread-block execution. When the complex multiplication step is finished, the resulting samples are brought back to the time domain by applying an inverse FFT in shared memory and aliased samples are removed before storing them to the device memory. This ensures high data reuse of both segment and filter samples.

[9] A thread-block is a set of GPU threads that execute the same code and can cooperate using shared memory.

ALGORITHM 2: Pseudo-code for the shared-memory OLS implementation. For input, we have input data x and set of filters f. The output is the convolved result y. The shared-memory FFT functions (`ForwardFFT` and `InverseFFT`) are either Cooley-Tukey C2C FFT for the complex input or Stockham FFT R2C and C2R, respectively, for the real input.

Input: x, f;
Output: y;
t = `threadId`;
b = `blockId`;
Forward FFT of the filters;
F = `ForwardFFT` (f);
Each thread-block process one segment;
GPU kernel begin
    Reading signal segment;
    $a[t] = x[bN_{\text{Seg}} + t]$;
    Forward FFT of the individual segments;
    A = `ForwardFFT` (a);
    Per-element complex multiplication of the segment a with F filters;
    for $r = 0$ **to** N_{fil} **do**
        A[t] = A[t] × F[r][t];
        a = `InverseFFT` (A);
        y = `RemoveAliasedSamples` (a);
    end
end

We have chosen different FFT algorithms for C2C and R2R OLS implementations. The Cooley-Tukey FFT algorithm is more suited to complex-to-complex convolutions, because we can use the fact that, for a point-wise frequency domain convolution, the order of the data elements in the convolved arrays does not matter as long as the order of the elements is the same for both the input signal segment and the filter, provided that the inverse FFT can work with the same order of elements. In normal circumstances, the Cooley-Tukey FFT algorithm requires a reordering to take place on the input or output data, but when used in convolution, we can forgo this step and save some execution time.

The Stockham FFT algorithm is used to facilitate real-to-complex and complex-to-real Fourier transformation [19] these require that the elements of the input and output of the FFT algorithm are in the correct order. The Stockham FFT algorithm is an auto-sort algorithm, which satisfies this condition. Our shared-memory implementation of the Stockham FFT algorithm is 30% slower on average than our shared-memory implementation of the Cooley-Tukey FFT algorithm without the reordering step. This performance penalty is redeemed by the fact that for real-to-complex and complex-to-real Fourier transformations, we can use an FFT length of half the size (compared to a C2C FFT) as described in Reference [19].

The benefit of having one GPU kernel is not only eliminating device memory accesses, but it also lowers memory requirements, because we do not need to store intermediate results as with the cuFFT-OLS implementation. The disadvantage of this approach is that it works well only for small filter sizes $M \lesssim 3{,}300$ (for Titan V GPU). This limitation is imposed by the size of the GPU shared memory.

Table 1. GPU Card Specifications

	P100	P4	TITAN V
CUDA Cores	3,584	2,560	5,120
SMs	56	20	80
Base/Max Core Clock	1,126/1,303 MHz	810/1,063 MHz	1,220/1,455 MHz
Memory Clock	1,406 MHz	6,000 MHz	850 MHz
Gl. m. bandwidth	720 GB/s	192 GB/s	652 GB/s
Shared m. bandwidth	9121 GB/s	2657 GB/s	14550 GB/s
Memory size	16 GB	8 GB	12 GB
TDP	250 W	75 W	250 W
Max. sh. memory per thread-block	48 kB	48 kB	48/96 kB

The shared-memory bandwidth is calculated as BW(bytes/s) = (bank bandwidth (bytes)) × (clock frequency (Hz)) × (32 banks) × (# multiprocessors). We have used CUDA version 10.0.130 and cuDNN version 7.5.0.

The analysis of the SM-OLS GPU kernel reveals that it is limited by the shared-memory bandwidth. For R2R version the kernel utilizes around 75% of the shared-memory bandwidth. The utilization is lower (50%) for a segment size of 4096 elements. For the C2C version, the bandwidth utilization of the shared-memory bandwidth is 50%. This is in part because, for the first few iterations in the FFT routine, we use shuffle instructions, which are not reflected by shared-memory bandwidth utilization. The use of the shuffle instructions, however, increases utilization of the load-store instruction, which is also high. The floating point (FP32) compute utilization is also high. The occupancy, a ratio of the maximum amount of active threads per streaming multiprocessor (SM) and active threads per SM, is only 50%. This is a consequence of high register count used by the convolution kernel. The GPU registers are used to store the signal segment elements after forward Fourier transform, which is reused, and they are also used for storage of the currently processed signal segment, which undergoes inverse Fourier transformation. The device memory bandwidth utilization ranges from 60% down to 25% for longer signal segments. The situation is similar for GPU kernels with non-local post-processing.

4 RESULTS

For our investigation, we have used three NVIDIA GPU cards, the P100 GPU, the P4 GPU, and the TitanV GPU (hardware specifications can be found in Table 1).

We have compared both shared-memory implementations (C2C, R2R) of OLS convolution (SM-OLS) for several different filter and signal lengths and also for a varying number of filters with convolution implementations based on the cuDNN library and our implementation of the OLS method, which uses cuFFT (cuFFT-OLS). For our results presented here, we have chosen to limit the input signal length to 2 million points or the number of filters to 8 (unless otherwise stated), to include the P4 GPU in our comparisons. The reason for this is that the P4 GPU card has a smaller device memory capacity and as such cannot process the same problem sizes as the P100 GPU or TitanV GPU.

The length of the input signal or filter length, as well as the number of filters in our implementation, can be arbitrary and they are not limited to presented values. We have chosen the value of these quantities to present the scaling behaviour of the problem. The input signal length is arbitrary from the nature of the OLS method and limited only by available memory. The filter length is arbitrary, but in the case of the shared-memory OLS it is limited by the maximum size of the FFT that can be processed (currently $N = 4{,}096$ points). The number of filters is also arbitrary and limited only by available memory.

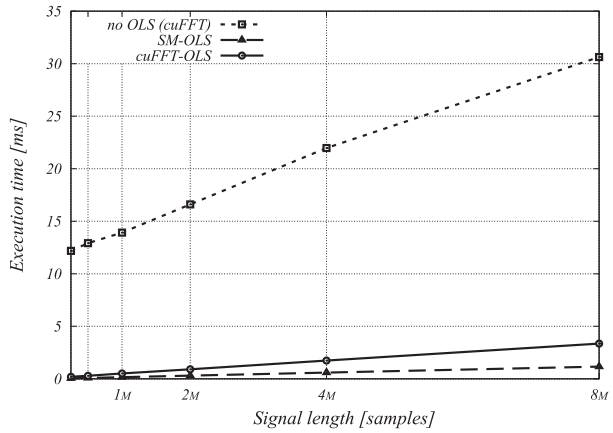

Fig. 3. Comparison of the execution time of convolution without OLS method using cuFFT, convolution via OLS method using cuFFT and convolution via custom FFT in shared memory. Results are for 8 filters of length 64 on TITAN V.

First, we have compared convolution without the OLS method using cuFFT, although OLS is a well established method this comparison shows how ineffective the standard convolution through the frequency domain can be for the case of convolution with multiple small filters. The execution time for convolution without OLS is presented in Figure 3.

4.1 Comparison with cuDNN Library Convolution

We begin by comparing the one-dimensional real-to-real SM-OLS convolution with one-dimensional real-to-real convolution via the cuDNN library. The execution time for the different input signal lengths and for the different number of filters is shown in Figure 4. The speedup factor for the same configurations is shown in Figure 5.

4.2 Comparison with cuFFT OLS Convolution

Next, we present results for the comparison of complex-to-complex (C2C) Fourier domain convolution implementations. The execution time and the number of processed elements versus the number of filters, and versus input signal length is presented in Figure 6.

The speed-up factors for different filter lengths versus the number of filters and versus the input signal length used are presented in Figure 7. Furthermore, speedups for signal length other then 2M samples are shown in Figure 8.

The cuFFT-OLS convolution performs best with segment size $N = 8{,}192$ for most of the filter sizes that we have investigated. The best performing segment size in the case of the SM-OLS convolution varies, this is because our FFT implementation performs better for smaller FFT lengths. Figure 9 shows how the performance of the SM-OLS convolution depends on the chosen FFT length (for TitanV GPU). Smaller FFT sizes become less effective with longer filter lengths, because the aliased part of the segment becomes a higher fraction of the overall FFT size and more segments are necessary to calculate the OLS convolution.

The comparison of real-to-real SM-OLS with cuFFT-OLS is similar. The execution time and the number of elements processed per second versus the number of filters, and versus input signal length is shown in Figure 10.

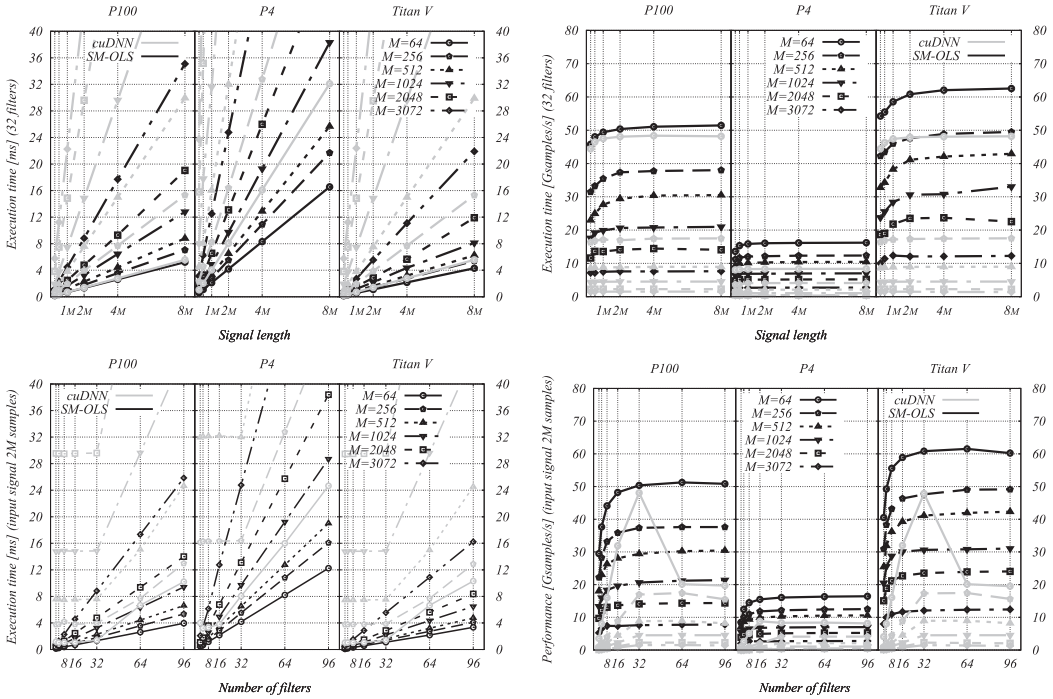

Fig. 4. The execution time of the R2R convolution on the left and the number of elements processed per second on the right-hand side via cuDNN (gray) and shared-memory OLS (black) for different input signal lengths (top) and different number of filters (bottom).

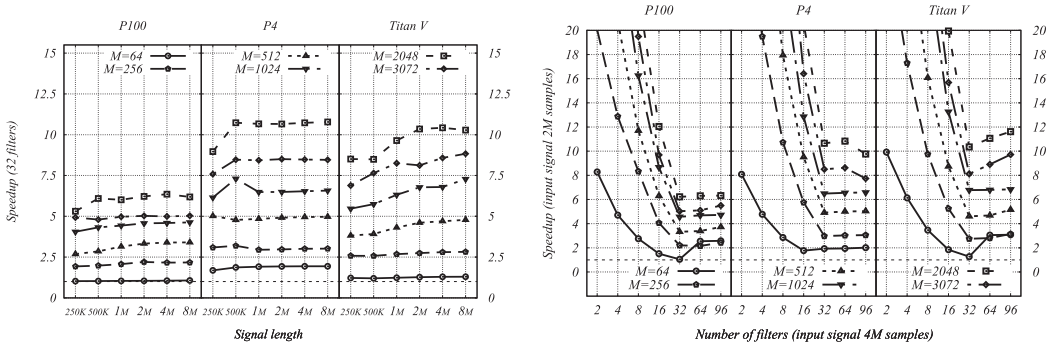

Fig. 5. The speedup factors of the R2R SM-OLS convolution with respect to the cuDNN convolution for different input signal lengths (left) and different number of filters (right).

The speed-up factors for different filter lengths versus the number of filters and versus the input signal length used for 2M signal length are presented in Figure 11. Speedups for signal lengths other than 2M samples are shown in Figure 12.

4.3 Non-local Post-processing

The advantage of the SM-OLS method is that it has access to all output elements of a given segment. This allows us to perform, in addition to per-element post-processing (for example, the

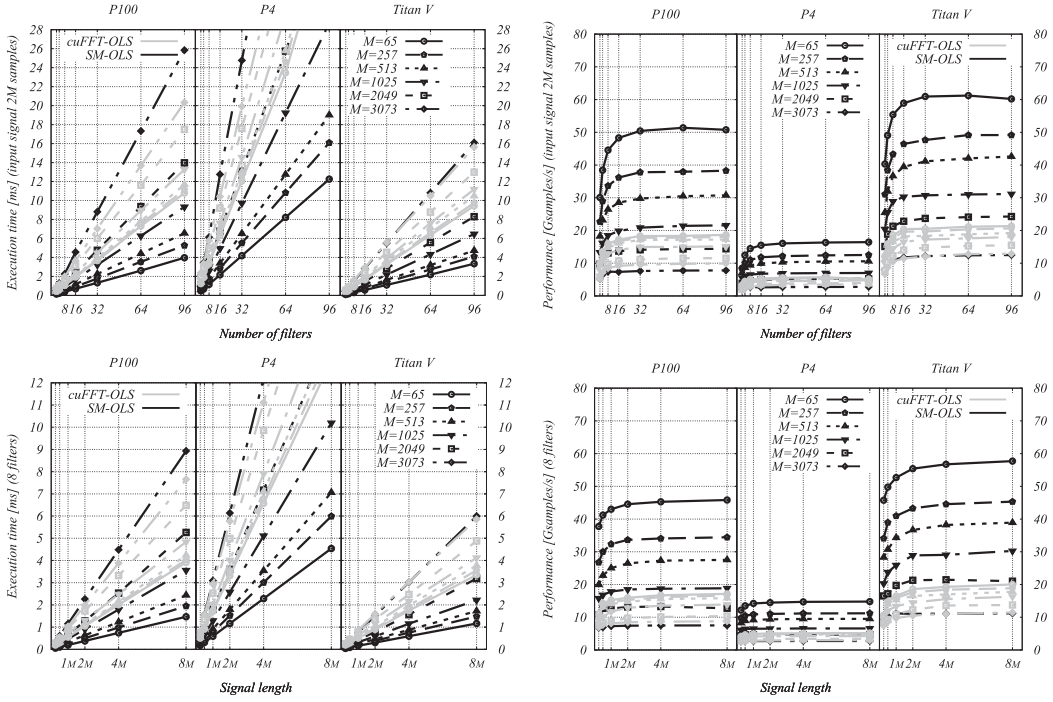

Fig. 6. The execution time of the C2C convolution on the left and the number of elements processed per second on the right-hand side of the SM-OLS convolution (black) and the cuFFT-OLS convolution (gray) for different number of filters (top) and increasing input signal length (bottom).

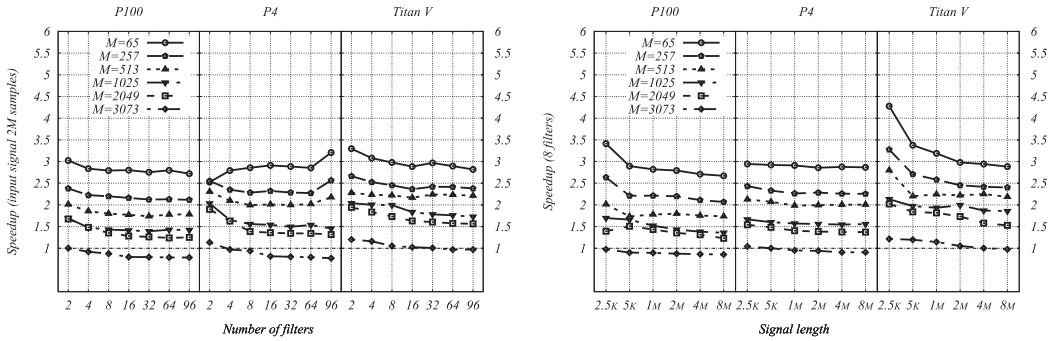

Fig. 7. The speed-up of the C2C SM-OLS convolution with respect to the C2C cuFFT-OLS convolution implementation for different filter lengths vs. the number of filters (left), and vs. the signal length (right).

calculation of the power spectrum), non-local post-processing as well (for example, the numerical derivative or interpolation). The non-local post-processing of output data requires access to the immediate or extended neighborhood of the element to be processed. The cuFFT-OLS method with callbacks offers only limited capabilities when an output element needs to access the values of neighboring elements. The achieved speedups with non-local post-processing are shown in Figure 13, where we have calculated the derivative of the convolved signal. We have chosen to calculate the derivative, because it does not require a larger memory footprint for the output,

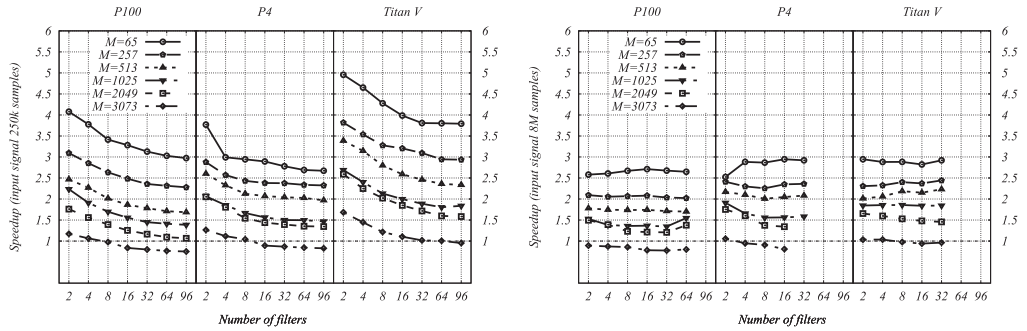

Fig. 8. The speed-up of the C2C SM-OLS convolution with respect to the C2C cuFFT-OLS convolution implementation for different filter lengths vs. the number of filters (left) for signal lengths 250k and 8M samples. The number of filters is limited by amount of device memory the GPU has, this is why there are missing points for P4 GPU (8 GB) and TitanV GPU (12 GB).

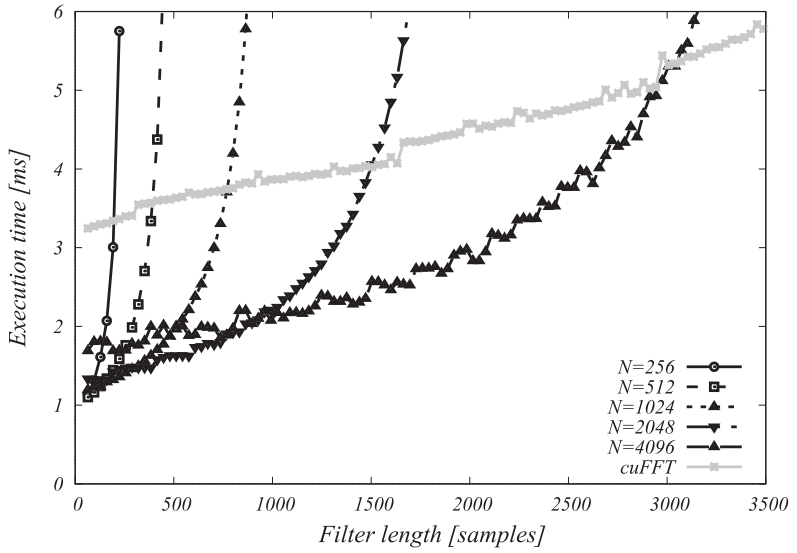

Fig. 9. The execution time of the SM-OLS convolution vs. filter length for different segment (FFT) sizes. The execution time of the cuFFT-OLS convolution is added for comparison.

thus the amount of data that needs to be transferred to and back from device memory remains the same.

4.4 PCI-e Latencies

The SM-OLS convolution implementation presented here is most efficient when used as a part of larger signal processing/data reduction pipeline. If run independently, then the execution time, which includes PCI-e transfer times, would be dominated by the time taken to transfer the output data to the host. Further processing of the output data from the convolution output (such as peak finding or candidate selection) would reduce the amount of output data transferred to the host to a point where the transfer of the output data could be hidden by the computations.[10]

[10]Using, for example, CUDA Streams.

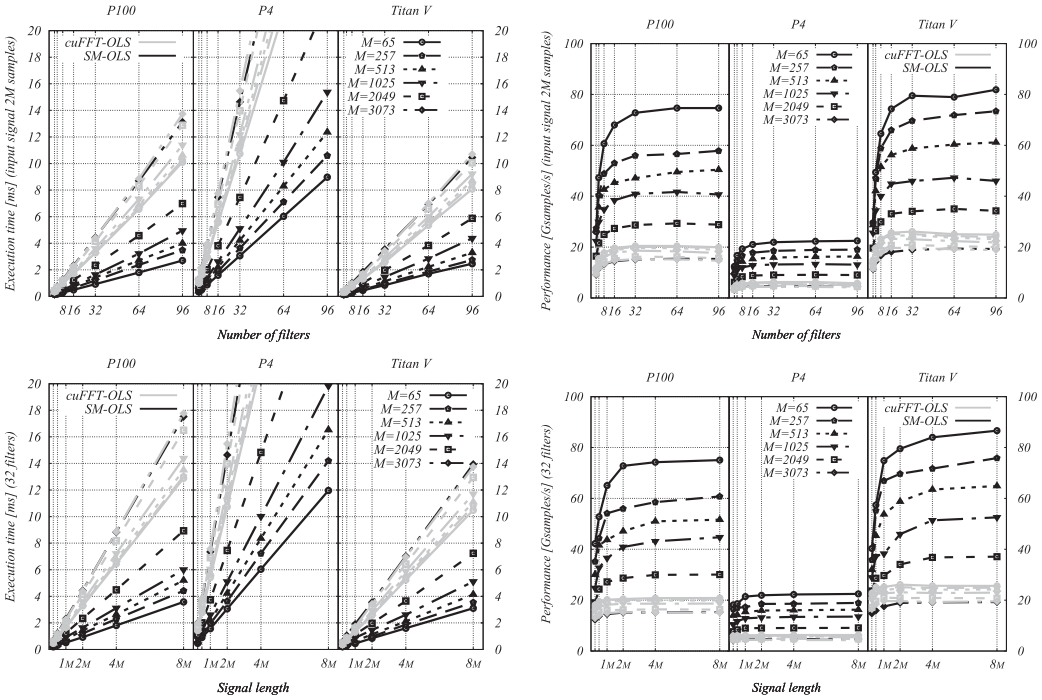

Fig. 10. The execution time of the R2R convolution on the left and the number of elements processed per second on the right-hand side of the SM-OLS convolution (black) and the cuFFT-OLS convolution (gray) for different number of filters (top) and increasing input signal length (bottom).

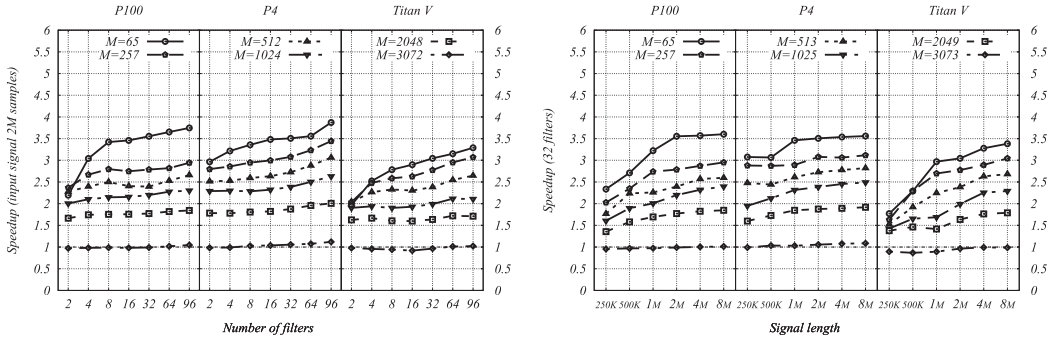

Fig. 11. The speed-up of the R2R SM-OLS convolution with respect to the R2R cuFFT-OLS convolution implementation for different filter lengths vs. the number of filters (left), and vs. the signal length (right).

5 DISCUSSION

The main source of the speedup for our shared-memory OLS implementation for one-dimensional convolution is the elimination of device memory accesses during the convolution step in the OLS method. If every other aspect of the computations in SM-OLS and cuFFT-OLS were equal, then the elimination of device memory accesses would result in a constant speedup for all filter lengths, the number of filters or signal length, since the only difference between the two cases would be

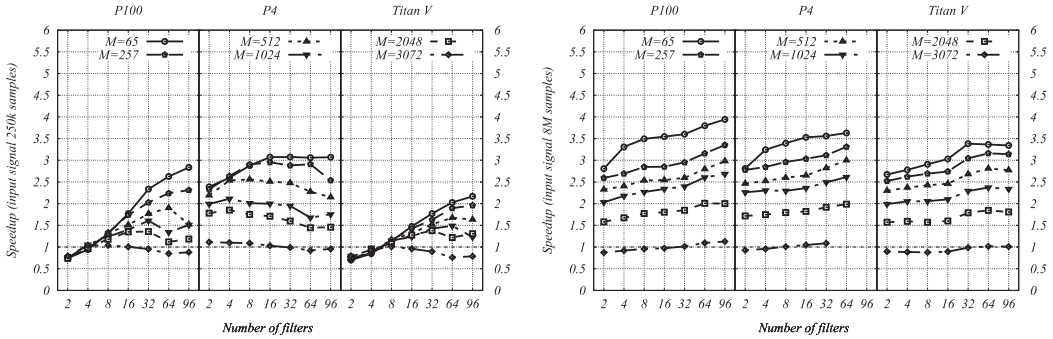

Fig. 12. The speed-up of the R2R SM-OLS convolution with respect to the R2R cuFFT-OLS convolution implementation for different filter lengths vs. the number of filters (left) for signal lengths 250k and 8M samples. The number of filters is limited by the amount of device memory the GPU has, this is why there are missing points for P4 GPU (8 GB) and TitanV GPU (12 GB).

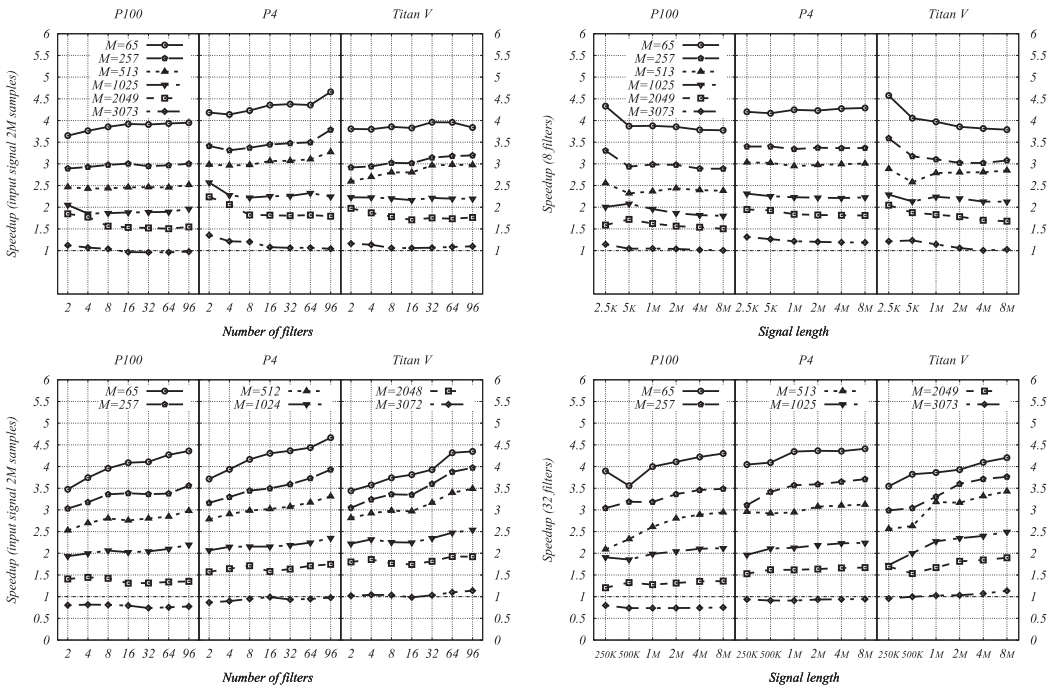

Fig. 13. The speed-up of the SM-OLS over cuFFT-OLS when non-local post-processing is included into consideration. The speed-up for C2C convolution is at the top and speed-up for R2R convolution is at the bottom.

the per-sample device memory accesses that were not realised. In real calculations, there are many other effects which affect the speedup of our shared-memory implementation of OLS convolution.

The primary effect is determined by the segment size N, which needs to be set appropriately so that the number of aliased samples which are given by the filter length M is proportionally small compared to the number of uncontaminated output samples contained in the output segment. The segment size in the cuFFT-OLS convolution implementation is not limited to any particular size. This is not true for our implementation of the SM-OLS convolution. Our SM-OLS is limited to a

segment length of 4,096 samples. This limitation is imposed by the size of the shared memory and also by the number of samples we are able to process per thread.

If we fix the segment size N, then any increase in filter length leads to a decrease in the number of correct output samples per segment, thus more segments are required to calculate the whole convolution. The effect of this can be observed in Figure 9 where different black lines represent the execution time of the SM-OLS implementation with a fixed segment (FFT) size. Figure 9 shows that each segment size is optimal only for a limited range of filter lengths and after that, it is better to switch to a different segment size. Since our implementation of the SM-OLS convolution is limited to segment size $N = 4{,}096$, we cannot use a longer segment size when the number of correct samples per segment decreases below a certain limit and at that point cuFFT-OLS becomes the better performing implementation.

The caching of filters is also governed by the size of the segment. The filter length in the frequency domain is equal to the size of the segment N, so by increasing segment size, we are decreasing the number of filters which can be cached by the GPU's fixed size cache at any instant.

5.1 Comparison with cuDNN Convolution

Figure 4 shows the comparison of the execution time of our SM-OLS convolution implementation and our implementation of convolution via cuDNN library. The execution time scales linearly with the input signal length shown on the left-hand side of Figure 4. Different scaling can be seen as the number of filters increase. Our implementation of SM-OLS scales linearly, but cuDNN has the same execution time up until the number of filters reaches 32, at which point it scales linearly. This is due to under-utilization of the GPU resources, which is most probably caused by different work distribution, which favours more filters.

Figure 5 shows speedup factors of SM-OLS convolution implementation over the cuDNN convolution. The speedup factors for different filter lengths (different line types) versus the signal length (on the left-hand side of Figure 5) shows that both implementations scale at the same rate as the signal length increases. Figure 5 indicates that the cuDNN library is optimised for small filter lengths, since convolutions with smaller filters have lower speedups. The reverse is true when SM-OLS convolution is compared to cuFFT-OLS convolution. The high speedups shown on the right-hand side of Figure 5 are due to poor scaling of the cuDNN library for a number of filters below 32.

5.2 Comparison with cuFFT Convolution

The execution time, as shown for C2C convolutions in Figure 6 and for R2R convolutions in Figure 10, scales linearly with an increasing number of filters and increasing input signal length. Both implementations achieve roughly constant performance in the number of processed elements per second past 16 filters or a signal length of two million samples.

The speedup factors of SM-OLS convolution over cuFFT-OLS convolution are shown for C2C convolutions in Figures 7 and 8 and for R2R convolutions in Figures 11 and 12. The speedup factors are, in the majority of cases, constant and do not change with the number of filters or the length of the input signal. This is because the segment size is not affected by these parameters, the only difference between the two implementations is the number of device memory accesses performed, or rather not-performed, per sample by the SM-OLS implementation. The total number of processed samples, which includes also the aliased samples, might be different between the two implementations due to different segment sizes used, but the ratio of device memory transfers between these two implementations of the OLS method remains constant and as such the speedup remains constant as well.

There are exceptions to this rule. In the case of complex-to-complex convolutions, we observe (in Figure 7 and on the left-hand side of Figure 8) that for a small number of filters or short signal lengths, we have higher speedups.

The higher speedup for short signal lengths is due to the slower performance of the cuFFT-OLS convolution, which under-utilises GPU resources in this regime. The cuFFT-OLS performs best with longer segment sizes (8,192), which, for shorter signal lengths, does not provide enough parallelism for the GPU to utilise. The Titan V GPU, which has the most SM,[11] has the highest speedups, while P4 GPU, which has the fewest SMs, is barely affected.

The high speedups for the small filter numbers are caused by the overhead of creating segments in the cuFFT-OLS implementation. This step is, in the case of SM-OLS, included in the GPU kernel and does not create additional device memory accesses.

The situation is different for R2R convolutions. Speedup factors of SM-OLS convolution over cuFFT-OLS convolution are shown in Figures 11 and 12. We see that for cases with short signal lengths the SM-OLS achieves low or below one speedups. This is caused by the under-utilisation of the GPU resources in our SM-OLS implementation. In the case of R2R convolutions, we are able to convolve a segment of size N with an FFT size $N/2$ [19], meaning that we are able to fit (depending on the FFT size) up to four thread-blocks per SM, which leads to under-utilization, even for signal sizes of 500k samples. This can be best observed in Figure 12 on the left-hand side, where we show speedups for short signal lengths (250k). GPU cards that are most affected (TitanV GPU, P100 GPU) have also the most SMs, while the P4 GPU with a smaller number of SMs shows speedups comparable to what we can see in Figure 11.

Last, our SM-OLS has lower performance for shorter filters. This is due to shared-memory bank conflicts in our shared-memory implementation of the Stockham FFT algorithm. These shared-memory bank conflicts occur in the first few iterations of the algorithm. The execution time of these first few iterations dominates the execution time of the shorter FFTs and thus decreases the performance of the whole convolution.

5.3 Non-local Post-processing

Figure 13 shows the speedup of SM-OLS over cuFFT-OLS when performing a non-local post-processing step. Examples, where this might be required, include interpolation of the output or numerical differentiation (which we have used to demonstrate this). The change in the performance depends on the filter size used. The speedup can also decrease when compared to convolution without non-local post-processing. This can be seen for P100 and P4 GPUs when performing real-to-real convolutions with filters longer than 1,025 samples, but for shorter filter lengths the speedup can be as great as 30% as in the case of the Titan V GPU for filter lengths 257 and 513.

6 CONCLUSIONS

We have presented an implementation of the shared-memory overlap-and-save method for the one-dimensional convolution of a large data set with a set of short filters. We have demonstrated a significant speed-up for our shared-memory implementation of overlap-and-save, over an implementation of the overlap-and-save method that uses a vendor-supplied FFT library (cuFFT). We have also demonstrated a speedup in the calculation of convolution over a vendor-supplied library for deep neural network primitives (cuDNN) for NVIDIA GPUs. This work has been used to enable real-time data processing in AstroAccelerate software package [24] that performs the Fourier Domain Acceleration Search for the Square Kilometre Array [1, 4, 5]. Considering the significance of

[11] The SM or streaming multiprocessor is a set of computing cores, the exact number of cores depends on the architecture, which executes threads instruction in parallel.

convolution in signal processing this implementation could have a noticeable impact in fields such as natural language processing, monitoring and listening services, speech recognition or pattern matching.

Future work includes the incorporation of the shared-memory FFT presented in this article, into our implementation of a polyphase filter [2] to increase its data throughput.

ACKNOWLEDGMENT

The authors acknowledge the use of the University of Oxford Advanced Research Computing (ARC) [20] facility in carrying out this work.

REFERENCES

[1] K. Adámek, S. Dimoudi, M. Giles, and W. Armour. 2017. Improved acceleration of the GPU fourier domain acceleration search algorithm. In *Proceedings of the 27th Astronomical Data Analysis Software and Systems Conference (ADASS'17)*. arxiv:astro-ph.IM/1711.10855

[2] K. Adámek, J. Novotný, and W. Armour. 2016. A polyphase filter for many-core architectures. *Astron. Comput.* 16 (July 2016), 1–16. DOI: https://doi.org/10.1016/j.ascom.2016.03.003 arxiv:astro-ph.IM/1511.03599

[3] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch. 1967. What is the fast Fourier transform?*Proc. IEEE* 55, 10 (Oct. 1967), 1664–1674. DOI: https://doi.org/10.1109/PROC.1967.5957

[4] S. Dimoudi, K. Adamek, P. Thiagaraj, S. M. Ransom, A. Karastergiou, and W. Armour. 2018. A GPU implementation of the correlation technique for real-time Fourier domain pulsar acceleration searches. *The Astrophysical Journal Supplement Series* 239, 2 (2018). DOI: 10.3847/1538-4365/aabe88

[5] S. Dimoudi and W. Armour. 2015. Pulsar acceleration searches on the GPU for the square kilometre array. In *Proceedings of the 25th Astronomical Data Analysis Software and Systems Conference (ADASS'15)*. arxiv:astro-ph.IM/1511.07343.

[6] T. Dobashi and H. Kiya. 2013. A parallel implementation method of FFT-based full-search block matching algorithms. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. 2644–2648. DOI: https://doi.org/10.1109/ICASSP.2013.6638135

[7] J. A. Fernandez and B. V. K. V. Kumar. 2013. Multidimensional overlap-add and overlap-save for correlation and convolution. In *Proceedings of the IEEE International Conference on Image Processing*. 509–513. DOI: https://doi.org/10.1109/ICIP.2013.6738105

[8] Jeremy Fowers, Greg Brown, John Wernsing, and Greg Stitt. 2013. A performance and energy comparison of convolution on GPUs, FPGAs, and multicore processors. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013). DOI: https://doi.org/10.1145/2400682.2400684

[9] Naga Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. 2008. High performance discrete fourier transforms on graphics processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. Retrieved from https://www.microsoft.com/en-us/research/publication/high-performance-discrete-fourier-transforms-on-graphics-processors/.

[10] Eladio Gutierrez, Sergio Romero, Maria A. Trenas, and Emilio L. Zapata. 2008. *Memory Locality Exploitation Strategies for FFT on the CUDA Architecture*. Springer-Verlag, Berlin, 430–443. https://doi.org/10.1007/978-3-540-92859-1_39

[11] John W. Tukey and James W. Cooley. 1965. An algorithm for the machine calculation of complex fourier series. *Math. Comp.* 19, 90 (1965), 297–301. Retrieved from http://www.jstor.org/stable/2003354.

[12] M. Jordà, P. Valero-Lara, and A. J. Peña. 2019. Performance evaluation of cuDNN convolution algorithms on NVIDIA volta GPUs. *IEEE Access* 7 (2019), 70461–70473. DOI: https://doi.org/10.1109/ACCESS.2019.2918851

[13] A. Lavin and S. Gray. 2015. Fast algorithms for convolutional neural networks. *ArXiv e-prints* arxiv:1509.09308.

[14] R. G. Lyons. 2011. *Understanding Digital Signal Processing*. Prentice Hall.

[15] Kenneth Moreland and Edward Angel. 2003. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'03)*. Eurographics Association, Aire-la-Ville, Switzerland, 112–119. Retrieved from http://dl.acm.org/citation.cfm?id=844174.844191.

[16] M. J. Narasimha. 2006. Modified overlap-add and overlap-save convolution algorithms for real signals. *IEEE Signal Process. Lett.* 13, 11 (Nov. 2006), 669–671. DOI: https://doi.org/10.1109/LSP.2006.879475

[17] NVIDIA. 2019. NVIDIA CUDA Deep Neural Network Library (cuDNN). Retrieved from https://developer.nvidia.com/cudnn.

[18] NVIDIA. 2019. NVIDIA CUDA Fast Fourier Transform Library (cuFFT). Retrieved from https://developer.nvidia.com/cufft.

[19] W. H. Press. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press.
[20] Andrew Richards. 2015. *University of Oxford Advanced Research Computing*. DOI: https://doi.org/10.5281/zenodo.22558
[21] C. Van Loan. 1992. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics. Retrieved from arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9781611970999.
[22] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. *ArXiv e-prints* arxiv:cs.LG/1412.7580. https://research.fb.com/wp-content/uploads/2016/11/fast-convolutional-nets-with-fbfft-a-gpu-performance-evaluation.pdf.
[23] Vasily Volkov and Brian Kazian. 2008. Fitting FFT onto the G80 architecture. University of California, Berkeley (2008). https://pdfs.semanticscholar.org/eb3a/82ddfc4e73de18a4004ecb9c1109730ae3eb.pdf.
[24] W. Armour, K. Aámek, J. Novotný, S. Dimoudi, C. Carels, and N. Ouannoughi. 2019. AstroAccelerate. https://github.com/AstroAccelerateOrg/astro-accelerate.git.
[25] Frank Wefers and Michael Vorländer. 2013. Using fast convolution for FIR filtering Overview and guidelines for real-time audio rendering. In *Proceedings of the International Conference on Acoustics (AIA-DAGA'13)*. Retrieved from http://pub.dega-akustik.de/AIA_DAGA_2013/data/articles/000683.pdf.
[26] Yi Yang and Huiyang Zhou. 2014. *A Highly Efficient FFT Using Shared-memory Multiplexing*. Springer International Publishing, Cham, 363–377. DOI: https://doi.org/10.1007/978-3-319-06548-9_17

Received October 2019; revised March 2020; accepted April 2020